The MLCA: A Solution Paradigm for Parallel Programmable SoCs

(Invited Paper)

Tarek Abdelrahman, Ahmed Abdelkhalek, Utku Aydonat, Davor Capalija, David Han, Ivan Matosevic, Kirk Stewart
Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
Email:{tsa,abdel,uaydonat,davor,han, imatos,stewar}@eecg.toronto.edu

Faraydon Karim, Alain Mellan
AST
STMicroelectronics
4690 Executive Drive, Suite 200
San Diego, CA
Email: {faraydon.karim,alain.mellan}@st.com

***Abstract*— Parallel Programmable Systems-on-a-chip (PP-SoC) are quickly becoming the de facto architecture for high-performance embedded systems. The programming of these systems is a challenge that often increases the cost of system development. The Multi-Level Computing Architecture (MLCA) promises to address this programmability challenge by supporting a superscalar coarse-grain parallel programming model. In this paper, we describe the MLCA and its programming model. We provide an overview of the compilation environment we are developing for this architecture. We present an evaluation of the MLCA and its compiler support using realistic multimedia applications on a simulator and on an FPGA prototype of the MLCA. The results indicate the viability of this architecture for multimedia applications.**

I. Introduction

A de facto architecture that has emerged in the embedded systems market is that of a parallel programmable System-on-a-Chip (PP-SoC): an integrated design that incorporates into a single chip multiple programmable cores and various custom or semi-custom blocks and memories. This paradigm allows the reuse of pre-designed cores, thus amortizing the design cost of a core over many system generations. Today, there exists several commercial PP-SoCs, including ones by Daytona [1], [2], picoChip [3], Philips [4], [5] and Cradle Technologies [6].

However, the programming of PP-SoCs remains a challenge. Programming systems with multiple processors poses tasks (e.g., partitioning, synchronization, and communication) that are more complex and time-consuming than ones posed by single-processor systems. These tasks increase the time and cost of software development for PP-SoC applications. Although traditional parallel programming models, such as OpenMP and MPI facilitate the development of parallel programs, this activity remain difficult.

The *Multi-Level Computing Architecture* (MLCA) [7] is a novel architecture for PP-SoCs which contains support for the extraction and exploitation of parallelism among coarse-grain units of computations, or *tasks*. The novelty of the architecture stems from the fact that it does so using the same techniques used in today's superscalar processors to exploit instruction-level parallelism, such as register renaming and out-of-order execution. The resulting programming model is an intuitive coarse-grain data-flow model that is close to sequential programming: programmers do not have to explicitly specify data communication or synchronization; parallelism is automatically extracted by the underlying architecture. We believe that this architecture addresses the most difficult problem with today's PP-SoCs: *programmability*.

In this paper, we describe the MLCA architecture and its programming model. We describe the compiler support that is needed for this unique architecture. We give an experimental evaluation of the architecture using realistic multimedia applications and a prototype of the MLCA compiler. We conduct this evaluation on both a simulator of the MLCA and on an FPGA prototype implementation of the architecture. The results indicate that scaling performance can be obtained for realistic applications using the prototype compiler.

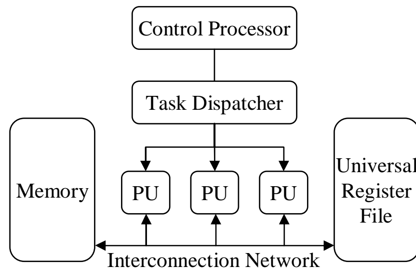
The remainder of this paper is organized as follows. Section II describes the MLCA and its programming model and constructs. Section III gives an overview of the MLCA compiler and the optimizations it performs. Section IV presents our experimental evaluation of the architecture. Finally, Section V gives some concluding remarks.

II. The MLCA

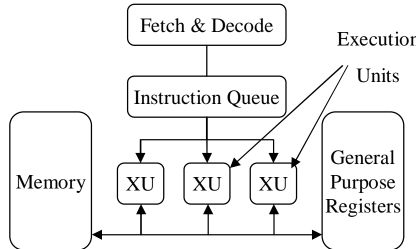
The MLCA [7] is a novel 2-level hierarchical architecture, aimed at parallel SoCs and primarily intended for multimedia applications. The lower level consists of multiple *processing units* (PUs), and the upper level of a controller that automatically exploits parallelism among coarse-grain units of computation, or *tasks*. A PU can be a full-fledged processor core, a DSP, a block of FPGA, or any other type of programmable hardware. The top-level controller consists of a *control processor* (CP), a *task dispatcher* (TD), and a *universal register file* (URF). A dedicated interconnection network links the PUs to the URF and to shared memory, as shown in Fig. 1(a).

The novelty of the MLCA stems from the fact that the upper level of the hierarchy supports parallel execution of tasks, using the same techniques used in superscalar processors, such as register renaming and out-of-order execution. This leverages existing processor technology to exploit task-level parallelism across PUs, in addition to possible instruction-level parallelism within each task. The similarity of the MLCA to the microarchitecture of a superscalar processor can be seen in Fig. 1.

1-4244-0417-7/06/$20.00 ©2006 IEEE



(a) MLCA.



(b) Superscalar processor.

Fig. 1. Comparison between the MLCA and a superscalar processor.

```
int Add() {
    int n1 = readArg(0);
    int n2 = readArg(1);

    writeArg(0, n1 + n2);

    return (n1+n2) != 0 ;
}
```

Fig. 2. An example task body.

```
do {
    ...
    notzero = Add(in width1, in width2,
                  out totwidth);
    notzero = Add(in width3, in totwidth,
                  out totwidth);
    if (notzero) {
        Div(in area1, in totwidth, out length1);
    }
    notzero = Add(in width4, in width5,
                  out totwidth)
    notzero = Add(in width6, in totwidth,
                  out totwidth);
    if (notzero) {
        Div(in area2, in totwidth, out length2);
    }
    ...
    notfinished = NotDone(in index);
} while(notfinished);
```

Fig. 3. The Sarek code of the example.

The upper level of the MLCA hierarchy may be implemented in hardware or in software. In the latter case, the software implements the register renaming and the out of order execution[1]. The decision to implement the upper level in hardware or software does not affect the programming model of the MLCA. Therefore, for the remainder of this paper, we make no assumptions about the implementation of this level.

The MLCA supports a programming model that, similar to sequential programming, does not require programmers to specify task synchronization and inter-task communication. It only requires programmers to express an application in terms of a sequential *control program*, which contains task instructions, and a set of *task functions*, each a sequential function with a specified number of input and output URF registers. In this paper, the task functions are assumed to be written in C, and the control program is expressed in a C-like language called *Sarek* [7].

Fig. 2 and Fig. 3 illustrate the MLCA programming model. The task `Add` shown in Fig. 2 is expressed as a `C` function that computes the sum of two integers. The function has no formal arguments. Instead, it communicates with the Sarek program through an API, obtaining input data with a *readArg* call, and writing results using an analogous *writeArg* call. Both calls specify the input and/or the output using a positional argument. Thus `readArg(1)` reads the second input to the function, while `writeArg(0)` writes the first output of the function. The task also returns a condition code that is written to a *condition register* in the CP, and may be used in a Sarek program to make control decisions, such as jumps or branches.

The main part of the corresponding Sarek program for the example is shown in Fig. 3. It makes four calls to the task `Add`. In each call, the variable names of the inputs and outputs of each task are specified. In addition, a direction indicator (`in` or `out`) is also specified for each variable. The second instance of the task `Add` must wait for the first instance to complete because of the true dependence caused by the use of the variable `totwidth`. However, the third and fourth instances of `Add` may proceed out-of-order with the first two even though they respectively write and read to `totwidth`, and also because there is no dependency with the conditional call to `Div`. The CP will automatically rename the register holding `totwidth` for these two tasks to eliminate the false output and anti dependences that exist.

[1]Indeed, a software implementation of the CP exists on a network of workstations [8].

III. Compilation For The MLCA

The overall compilation environment for the MLCA is shown in Fig. 4. It consists of the MLCA compiler, a performance estimator and a feedback generator. A sequential program is compiled by the MLCA compiler to generate optimized Sarek and task programs. The performance of these optimized programs is estimated using the performance estimator, and feedback is generated to the programmer indicating possible causes of performance bottlenecks. The programmer can then iteratively eliminate these bottlenecks. We believe this iterative development environment is particularly suitable for SoC application development. The remainder of this paper deals only with the MLCA compiler.

The MLCA compiler consists of two main components: a task generator and an optimizer. The goal of the task generator is to analyze the input sequential program and to produce an initial (unoptimized) control program and tasks. The programs are then processed by the optimizer to produce the optimized versions of these programs.

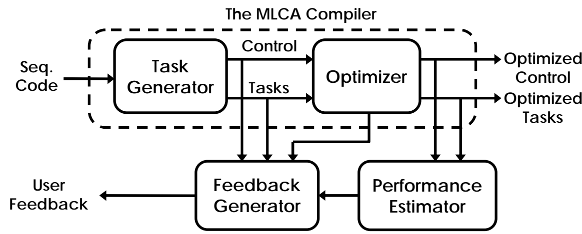


Fig. 4. The MLCA compilation environment.

The optimizer performs a number of code transformations that improve performance when tasks communicate through shared memory and that reduce power consumption.

A. Task Generation

Task generation is performed iteratively, starting from the main function of the input program. The control flow of the main function is converted to Sarek and becomes the control program. The computation performed by the main function is divided into multiple tasks at natural boundaries, such as loop bodies and function calls. The inter-task data flow is calculated, C variables are allocated to URF registers as necessary, and task argument lists are generated. The result is that functions called by main (first-level functions) are converted to tasks, and main has effectively been inlined into the control program. The performance optimizations are applied to the generated control program and task bodies, and the performance of the optimized program is evaluated. This process is repeated, using the task bodies as inputs, until successive iterations no longer show an improvement in performance.

Task generation is performed as a source-to-source transformation, producing a combination of Sarek and C code as output. This allows the programmer to examine the output and, if necessary, provide feedback to guide future iterations. The result is an environment in which a developer can rapidly experiment with different task granularities and the placement of task boundaries.

B. Performance Optimization

Tasks can communicate through the URF or through a shared memory. URF communication is desirable for scalar data (integers, floats, etc) because it enables the CP to eliminate false dependences by applying renaming. On the other hand, shared memory communication is more desirable for aggregate data (buffers and structures) because it minimizes communication overhead. In this case, a pointer to data in memory is communicated through the URF, allowing different tasks to access shared data using this pointer. However, shared memory communication can result in: (i) *renaming* and (ii) *synchronization* problems, both of which require compiler support.

The renaming problem is caused by the fact that the CP only renames URF registers, but not shared memory. Consequently, the CP can only eliminate the dependences caused by accesses to the pointers in URF, but not the actual data in memory. The synchronization problem results from the fact that dependences among tasks caused by accesses to memory are not necessarily reflected by dependences among pointers in task arguments. In other words, in order to access (read or write) data from shared memory, tasks only need to read a pointer from the URF, which does not introduce data dependence over the URF registers. This misleads the CP into issuing these tasks for parallel execution, possibly violating dependences caused by the accesses to shared data in memory.

We design code transformations [7], [9] in order to solve these renaming and synchronization problems and, thus, to improve performance. These transformations are *parameter deaggregation*, *buffer privatization* and *buffer renaming*. Parameter deaggregation aims to solve both problems. It achieves this by exposing the fields of structures in the parameter list of tasks, effectively transforming shared memory dependences into URF dependences. This enables the CP to enforce true dependences and to rename the fields of the structures. Buffer privatization aims to solve the renaming problem, specifically for buffers in memory. It effectively privatizes buffers for *tasks* whose accesses to these buffers create false dependences with the remaining tasks. In this sense, our buffer privatization transformation bears similarities with but also is different from array privatization [10], [11] which privatizes arrays for entire loop iterations. Buffer renaming addresses synchronization problems, by finding tasks that should not execute in parallel and introducing artificial URF dependences between them to ensure correct synchronization. It also eliminates such artificial dependences introduced by the programmer, if there are any.

C. Power Reduction

The optimizer also implements a profile-driven compiler technique for reducing the power consumption of MLCA applications using Dynamic Voltage Scaling (DVS) [12]. Our technique targets long-running loops in the control programs of MLCA applications. It combines analysis of the loop dependence graph with profiling information in order to deduce properties of the dynamic task graph that represents the run-time execution of tasks in the loop. Based on these deduced properties, our algorithm determines a suitable voltage level for the execution of each task. We achieve this goal using compact data structures and computationally lightweight procedures, rather than the explicit construction and analysis of the run-time task graph, which would be prohibitively expensive. Our technique includes a task scheduling algorithm that complements the voltage selection algorithm to ensure that power savings are achieved with little or no impact on the application execution time.

IV. Evaluation

We use two platforms to evaluate the MLCA. The first is a timed transaction-level simulator of the MLCA. It consists of approximately 6,000 lines of C++/SystemC, which reflect the overall structure of the MLCA, including a CP, TD, URF, ARM processors for PUs, and shared memory. The second platform is an FPGA prototype of the MLCA implemented on an Altera Stratix Nios II development board. In this prototype, 4 Nios II processors are used as PUs, and the CP and URF are implemented in software on a fifth processor. Each PU is clocked at 50 MHz and has an 8K instruction cache and an 8K data cache.

We use three realistic multimedia applications as benchmarks: MAD, an open source MPEG audio decoder, FMR, an open-source program that performs FM demodulation on

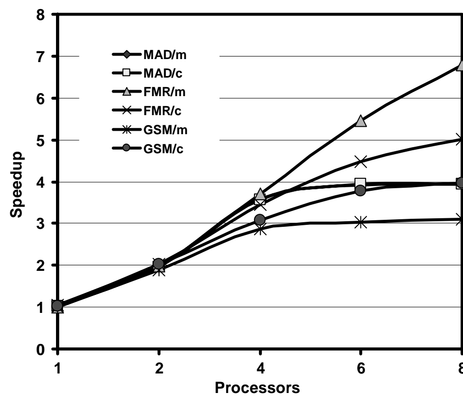


Fig. 5. Application speedups using the MLCA simulator.

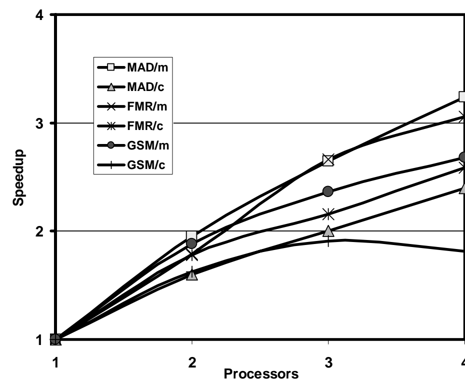


Fig. 6. Application speedups using the FPGA prototype.

Application	JPEG	GSM	MPEG
Energy saving	9.5%	5.6%	8.4%
Slowdown	0.5%	-0.3%	1.5%

TABLE I

PROCESSOR ENERGY SAVINGS AND EXECUTION SLOWDOWN

a 16-bit input data stream, and GSM, an open source implementation of the European GSM 06.10 standard for full-rate speech transcoding. These applications had already been manually ported and hand-optimized for the MLCA and we use these manually-ported versions as reference. In all our experiments, we run the MAD application to decode 64 frames of an audio file; FMR to decode 22 data packets and GSM to encode 64 frames of audio data.

We report the performance of the applications using the *speedup*, which is defined as the ratio of the execution cycles of the sequential program to the execution cycles of the parallel program with P processors. However, since it is not possible to run a purely sequential program on the MLCA, we report the speedup of a P-processor MLCA program relative to the parallel MLCA program running on a single processor.

At this time, the implementation of the task generator is still underway. Thus, we generate tasks manually following the process described in Section III until the set of tasks that exist in the manually ported versions are obtained. This provides us with un-optimized, but otherwise the same, versions of the applications. Given how the arguments to tasks are generated in the above process, it is natural that there is very little or no parallelism in these un-optimized versions.

Fig. 5 shows the speedups of the applications on the simulator. For each application, "/m" indicates the manually-ported version of the application, while "/c" indicates the compiler generated version. The figure shows that the speedups of the manual and compiler-generated versions are different, the compiler-generated versions scale with increasing processors in a manner similar to the manually-generated versions.

Fig. 6 shows the speedups of the applications on the FPGA prototype. The figure also shows that the performance of the compiler-generated versions scales similar to the manually-generated versions. The speedups on this platform are lower than on the simulator. We attribute this to bus and memory contention on the FPGA.

We evaluate our power-reduction techniques assuming that the power-related characteristics of the processors are identical to those of the Intel XScale processor [13]. In addition to the three applications above, we also use a JPEG application. Table I shows the savings in processor energy consumption achieved by the application of our technique, along with the incurred execution slowdown. The achieved power savings are significantly greater than those that could be achieved by uniformly slowing down all computations with only a similar increase in overall execution time.

V. CONCLUDING REMARKS

In this paper we presented an overview of the MLCA architecture and its compiler support. Experimental evaluation on both a simulator and an FPGA prototype of the MLCA using three realistic multimedia applications indicates that the architecture is viable in delivering scaling performance. Future work will focus on completing the task generation module and on further experimental evaluation using larger applications.

REFERENCES

[1] C. Nicol et. al, "A single-chip, 1.6-billion, 16-b MAC/s multiprocessor DSP," *IEEE J. of solid-state circuits*, vol. 35, no. 2, March 2000.

[2] A. Kalavade, J. Othmer, B. Ackland, and K. J. Singh, "Software environment for a multiprocessor DSP," in *Proc of the DAC*, pp. 827–83, 1999.

[3] PicoChip. `http://www.picochip.com`.

[4] M. Rutten, J. van Eijndhoven, and E. Pol, "Design of multi-tasking coprocessor control for eclipse," in *Proc. of Int'l Symp. on Hardware/Software Codesign*, pp. 139–144, 2002.

[5] S. Dutta, R. Jensen, and A. Rieckmann, "Viper: A multiprocessor SOC for advanced set-top box and digital TV systems," *IEEE Des. Test*, vol. 18, no. 5, pp. 21–31, 2001.

[6] "3SOC documentation - 3SOC 2003 hardware architecture, Cradle Technologies, inc., March 2002."

[7] F. Karim, A. Mellan, A. Nguyen, U. Aydonat, and T. Abdelrahman, "A multi-level computing architecture for multimedia applications," *IEEE Micro*, vol. 24, no. 3, pp. 55–66, 2004.

[8] A. Mellan and C. Stein, "Distributed hypercomputer," Tech. Rep. AST-SAN-2004-08, STMicroelectronics, 2004.

[9] U. Aydonat, "Compiler support for a multimedia system-on-chip architecture," Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 2005.

[10] P. Tu, *Automatic Array Privatization and Demand Driven Symbolic Analysis*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[11] Z. Li, "Array privatization for parallel execution of loops," in *Proc. of the ACM Int'l Conf. on Supercomputing*, pp. 313–322, ACM Press, 1992.

[12] I. Matosevic, T. S. Abdelrahman, F. Karim, and A. Mellan, "Power optimizations for the MLCA using dynamic voltage scaling," in *Proc. of SCOPES*, pp. 109–123, 2005.

[13] L. Clark et al., "An embedded 32-b microprocessor core for low-power and high-performance applications," *J. of Solid-State Circuits*, vol. 36, no. 11, pp. 1599–1608, 2001.